

Санкт-Петербургский политехнический университет Петра Великого  
Институт компьютерных наук и технологий  
Высшая школа интеллектуальных систем и суперкомпьютерных технологий

### **Отчёт по лабораторной работе № 3**

Дисциплина: Проектирование мобильных приложений  
Тема: Lifecycle компоненты. Навигация в приложении.

Выполнил студент гр. 3530901/90201 \_\_\_\_\_ Е.К. Борисов  
(подпись)

Принял старший преподаватель \_\_\_\_\_ А.Н. Кузнецов  
(подпись)

“ \_\_\_\_\_ ” \_\_\_\_\_ 2021 г.

Санкт-Петербург  
2021

## Оглавление

<b>Цели:</b> .....	<b>3</b>
<b>Задачи:</b> .....	<b>3</b>
Jetpack Compose .....	3
<b>Задание №1 ( codelab "Jetpack Compose basics")</b> .....	<b>5</b>
1.1.    Создание Compose проекта. ....	5
1.2.    Настройка пользовательского интерфейса.....	7
1.3.    Повторное использование Composable функций. ....	9
1.4.    Создание столбцов и строк. ....	10
1.5.    Состояния Compose.....	12
1.6.    Подъем состояния. ....	14
1.7.    Ленивые списки и rememberSaveable.....	17
1.8.    Анимация. ....	18
1.9.    Стилизация приложения. ....	19
<b>Задание №2</b> .....	<b>22</b>
<b>Задание №3</b> .....	<b>28</b>
<b>Задание №4</b> .....	<b>29</b>
<b>Задание №5</b> .....	<b>31</b>
<b>Вывод</b> .....	<b>36</b>
<b>Время на выполнение работы</b> .....	<b>36</b>

## **Цели:**

Познакомиться с Google Codelabs и научиться его использовать как способ быстрого изучения новых фреймворков и технологий.

Изучить основные возможности навигации внутри приложения: создание новых activity, navigation graph.

## **Задачи:**

1. Познакомьтесь с содержимым курса Jetpack Compose и выполнить codelab "Jetpack Compose basics"
2. Реализовать навигацию между экранами одного приложения согласно изображению ниже с помощью Activity, Intent и метода startActivityForResult.
3. Реализовать навигацию между экранами одного приложения согласно изображению ниже с помощью Activity, Intent и флагов Intent либо атрибутов Activity.
4. Дополнить граф навигации новым(-и) переходом(-ами) с целью демонстрации какого-нибудь атрибута Activity или флага Intent, который еще не использовался для решения задачи. Пояснить пример и работу флага/атрибута.
5. Решить исходную задачу с использованием navigation graph. Все Activity должны быть заменены на Fragment, кроме Activity 'About', которая должна остаться самостоятельной Activity.
6. Сравнить все решения.

## ***Jetpack Compose***

Jetpack Compose - это современный набор инструментов для создания собственного пользовательского интерфейса Android. Jetpack Compose упрощает и ускоряет разработку пользовательского интерфейса на Android с меньшим

количеством кода, мощными инструментами и интуитивно понятными API-интерфейсами Kotlin.

Jetpack Compose построен на основе составляемых функций. Эти функции позволяют программно определять пользовательский интерфейс вашего приложения, описывая его внешний вид и предоставляя зависимости данных, вместо того, чтобы сосредотачиваться на процессе создания пользовательского интерфейса (инициализация элемента, присоединение его к родительскому элементу и т. д.).

# Задание №1 ( codelab "Jetpack Compose basics")

## 1.1. Создание Compose проекта.

Android Studio предоставляет нам готовый шаблон проекта “Empty Compose Activity”, с которого мы начнем свою работу, следует также заметить, для использования Compose нужно использовать версию SDK не ниже 21 уровня API. После создания проекта мы видим следующий код в нашем главном Activity.

Листинг 1.1 MainActivity.kt

```
package com.example.basicscodelab

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.tooling.preview.Preview
import com.example.basicscodelab.ui.theme.BasicsCodelabTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                // A surface container using the 'background' color from the theme
                Surface(color = MaterialTheme.colors.background) {
                    Greeting("Android")
                }
            }
        }
    }

    @Composable
    fun Greeting(name: String) {
        Text(text = "Hello $name!")
    }

    @Preview(showBackground = true)
    @Composable
    fun DefaultPreview() {
        BasicsCodelabTheme {
            Greeting("Android")
        }
    }
}
```

Мы видим, что в отличие от обычного проекта где для определения макета мы используем setContentView с определением XML файла, здесь мы используем

`setContent`, в котором мы вызываем Composable функции. Внутри `setContent` мы видим тему приложения, которая позволяет задавать стили Composable функциям и которая зависит от названия проекта, имя темы можно изменить в файле `ui.theme/Theme.kt`.

Также у нас есть несколько Composable функций. Аннотация “`@Composable`” позволяет функции вызывать другие Composable функции внутри нее.

Функция `Greeting` выводит приветственное сообщение при запуске приложения. Внутри этой функции мы используем другую Composable функцию `Text`, которая является библиотечной.

Android Studio позволяет нам просматривать Composable функции внутри IDE, для этого используется аннотация “`@Preview`”, функция `DefaultPreview` как раз таки использует эту аннотацию и мы можем увидеть наше приветственное сообщение в окне предварительного просмотра.

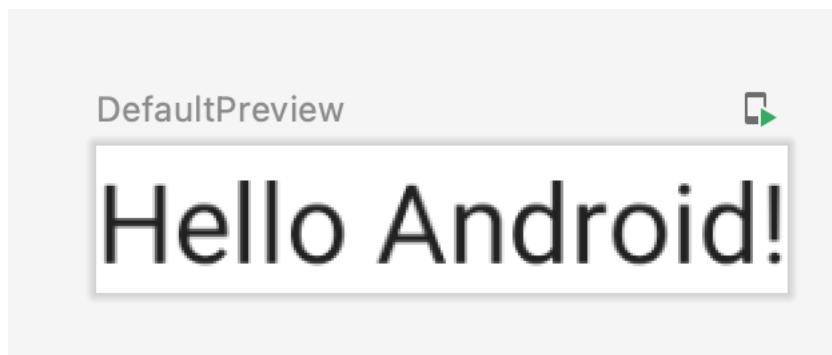


Рис. 1.1 Окно предварительного просмотра.

Возможно использовать несколько Preview в одном файле и задавать им имена.

## 1.2. Настройка пользовательского интерфейса.

В листинге 1.1 у нас также была функция Surface, эта функция позволяет задавать различные параметры для улучшения дизайна наших Composable функций. В качестве примера, зададим задний фон нашему приветственному сообщению.

Листинг 1.2.1 Использование Surface.

```
@Composable
fun Greeting(name: String) {
    Surface(color = MaterialTheme.colors.primary) {
        Text(text = "Hello $name!")
    }
}
```

Компоненты, вложенные в surface будут нарисованы поверх цвета этого фона.

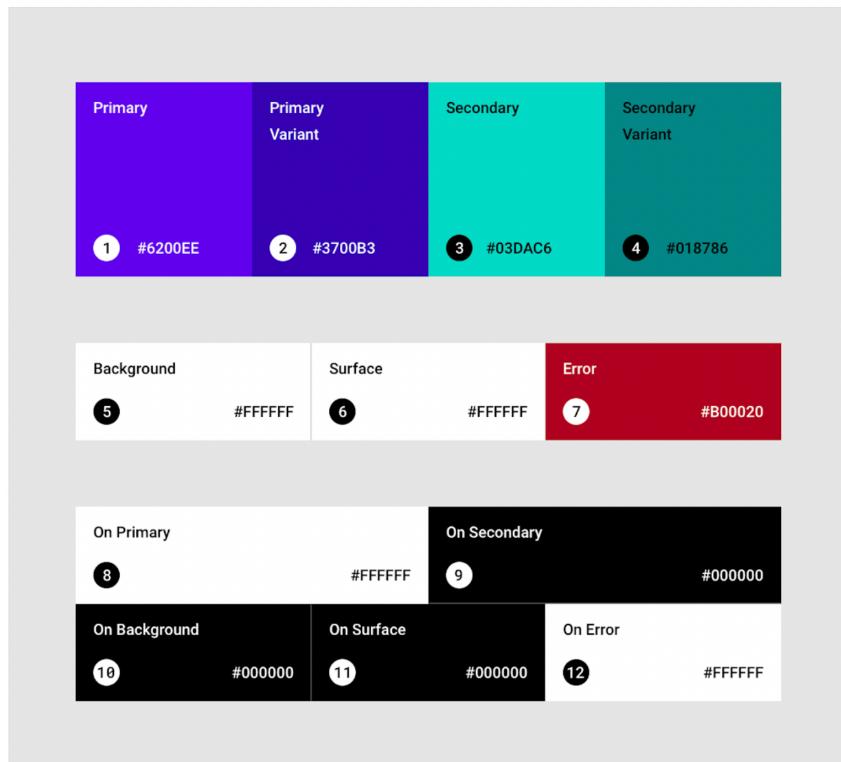


Рис. 1.2.1 Задний фон Greeting.

Наша функция Surface принимает в качестве параметра цвет, который задается при помощи MaterialTheme.colors. Surface и MaterialTheme являются концепциями, связанными с Material Design , системой дизайна, созданной Google. Цветовая система Material Design помогает осмысленно применять цвет к пользовательскому интерфейсу.

За счет использования MatherialTheme.colors нам не нужно думать о том, чтобы изменять цвет текста на белый, все это происходит по умолчанию, в данном случае у нас установлен primary цвет фона и любой текст, который находится внутри него должен использовать onPrimary цвет, который также определен в теме.

Все базовые цвета на рисунке ниже, можно настроить для нашего приложения.



## Модификаторы

Модификаторы сообщают элементу пользовательского интерфейса, как размещать, отображать или вести себя в его родительском макете. Модификаторы задаются в качестве параметра у большинства элементов пользовательского интерфейса, для примера добавим отступы для нашего сообщения Text.

Листинг 1.2.1 Использование модификаторов.

```
@Composable
fun Greeting(name: String) {
    Surface(color = MaterialTheme.colors.primary) {
        Text (text = "Hello $name!", modifier = Modifier.padding(24.dp))
    }
}
```



Рис. 1.2.2 Отступы Greeting.

### 1.3. Повторное использование Composable функций.

По мере создания приложения мы добавляем все больше и больше компонентов в пользовательский интерфейс и появляется большое количество уровней вложенности, чтобы улучшить читаемость кода и сделать отладку проще, следует создавать отдельные Composable функции, которые отвечают за отдельную часть экрана.

Листинг 1.3 Переиспользование функций.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp()
            }
        }
    }
}

@Composable
private fun MyApp() {
    Surface(color = MaterialTheme.colors.background) {
        Greeting("Android")
    }
}

@Composable
private fun Greeting(name: String) {
    Surface(color = MaterialTheme.colors.primary) {
        Text(text = "Hello $name!", modifier = Modifier.padding(24.dp))
    }
}

@Preview(showBackground = true)
@Composable
private fun DefaultPreview() {
    BasicsCodelabTheme {
        MyApp()
    }
}
```

Мы создали отдельную Composable функцию `MyApp()`, в которую поместили нашу функцию `Greeting` с аргументом “Android”, тем самым избежали дублирование кода в превью и коллбеке `onCreate`.

## 1.4. Создание столбцов и строк.

Jetpack Compose предоставляет три стандартных функции для задания расположения элементов интерфейса: `Column` – для вертикального расположения элементов, `Row` – для горизонтального и `Box` – для наложения элементов друг на друга.

Листинг 1.4 Расположение объектов.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp()
            }
        }
    }
}

@Composable
fun MyApp(names: List<String> = listOf("World", "Compose")) {
    Column {
        for (name in names) {
            Greeting(name = name)
        }
    }
}

@Composable
private fun Greeting(name: String) {
    Surface(
        color = MaterialTheme.colors.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier.weight(1f)) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            OutlinedButton(
                onClick = { /* TODO */ }
            ) {
                Text("Show less")
            }
        }
    }
}

@Preview(showBackground = true, widthDp = 320)
@Composable
private fun DefaultPreview() {
    BasicsCodelabTheme {
        MyApp()
    }
}
```

Мы можем использовать Composable функции, как и все функции в kotlin, мы сделали вывод двух сообщений Greeting с помощью цикла for и списка, которые отображаются на экране горизонтально благодаря функции Column. Также растянули окно предпросмотра и добавили отступы для каждого из сообщений с помощью модификаторов внутри Surface. Также был добавлен элемент OutLinedButton – это объект, который обворачивает текст, представляя его в виде кнопки. Эти кнопки отвечают за действия, которые важны, но не являются основными в приложении. Для того чтобы разместить кнопки справа, мы использовали модификатор weight для Column, тем самым текст заполнил все возможное доступное пространство, отодвинув кнопки вправо.

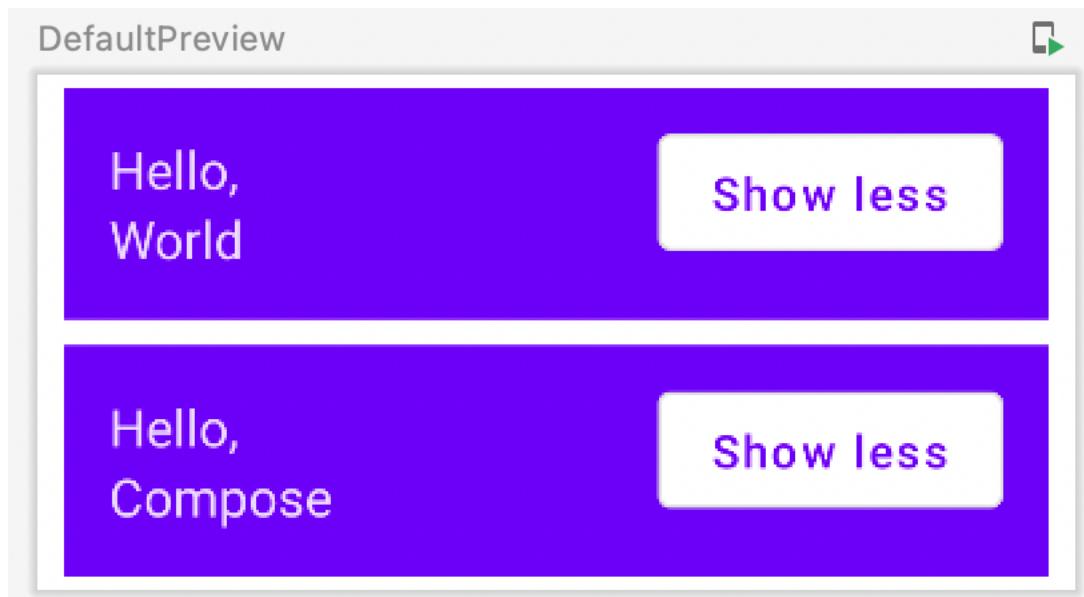


Рис. 1.4 Расположение элементов и кнопки.

## 1.5. Состояния Compose.

Добавим действия, которые будут совершаться при нажатии пользователем любой из наших кнопок. Для этого нам нужно отслеживать состояния каждого из этих кнопок, были ли они нажаты или нет. Если какие-то данные изменяются, Compose повторно вызывает эти функции с обновленными данными – это называется перекомпоновкой. Происходит перекомпоновка только в тех компонентах, данные в которых изменились и пропускает все остальные компоненты, для того чтобы отслеживать изменение, мы должны использовать `remember` и `mutableStateOf`.

Листинг 1.5 Отслеживание состояния.

```
@Composable
private fun Greeting(name: String) {

    val expanded = remember { mutableStateOf(false) }

    val extraPadding = if (expanded.value) 48.dp else 0.dp

    Surface(
        color = MaterialTheme.colors.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding))
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            OutlinedButton(
                onClick = { expanded.value = !expanded.value })
            ) {
                Text(if (expanded.value) "Show less" else "Show more")
            }
        }
    }
}
```

`MutableState` – это интерфейс, который содержит какое-либо значение и запускают перекомпоновку пользовательского интерфейса, когда это значение изменяется.

Remember – позволяет сохранить состояние компонента при перекомпоновке, таким образом при перекомпоновке значение не будет сброшено.

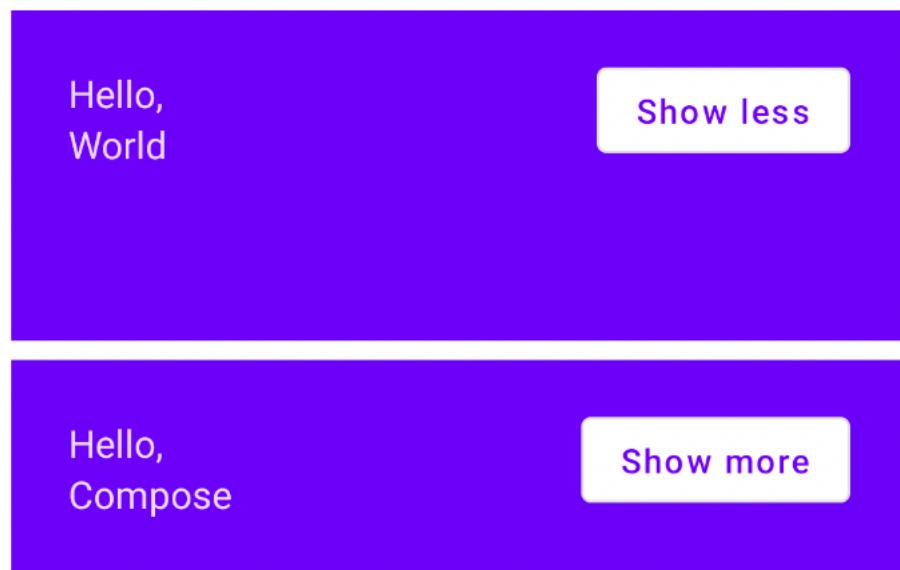


Рис. 1.5 Изменение состояния.

## 1.6. Подъем состояния.

В составных функциях состояние, которое читается или изменяется несколькими функциями, должно находиться в общем предке - этот процесс называется подъемом состояния. Допустим нам нужно отобразить некоторый стартовый экран приложения, который будет встречать пользователя и закрываться при нажатии на кнопку.

Листинг 1.6.1 Приветственный экран.

```
@Composable
fun MyApp() {
    if (shouldShowOnboarding) {
        OnboardingScreen()
    } else {
        Greeting()
    }
}

@Composable
fun OnboardingScreen() {
    // TODO: This state should be hoisted
    var shouldShowOnboarding by remember { mutableStateOf(true) }

    Surface {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text("Welcome to the Basics Codelab!")
            Button(
                modifier = Modifier.padding(vertical = 24.dp),
                onClick = { shouldShowOnboarding = false }
            ) {
                Text("Continue")
            }
        }
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodelabTheme {
        OnboardingScreen()
    }
}
```

Мы создали стартовый экран и превью для него, однако нам нужно как то узнать значение состояния, которое создано внутри функции OnBoardingScreen с функцией MyApp, для того чтобы она знала, какой экран нужно отображать пользователю.

Вместо того, чтобы каким-то образом делиться значением состоянием с его родителем, мы поднимаем состояние - мы просто перемещаем его к общему предку, которому требуется доступ к нему.

Листинг 1.6.2 Передача коллбэков вниз.

```
@Composable
fun MyApp() {
    var shouldShowOnboarding by remember { mutableStateOf(true) }

    if (shouldShowOnboarding) {
        OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
    } else {
        Greetings()
    }
}

@Composable
fun OnboardingScreen(onContinueClicked: () -> Unit) {
    Surface {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Text("Welcome to the Basics Codelab!")
            Button(
                modifier = Modifier.padding(vertical = 24.dp),
                onClick = onContinueClicked
            ) {
                Text("Continue")
            }
        }
    }
}

@Composable
private fun Greetings(names: List<String> = listOf("World", "Compose")) {
    Column(modifier = Modifier.padding(vertical = 4.dp)) {
        for (name in names) {
            Greeting(name = name)
        }
    }
}

@Preview(showBackground = true, widthDp = 320, heightDp = 320)
@Composable
fun OnboardingPreview() {
    BasicsCodeLabTheme {
        OnboardingScreen(onContinueClicked = {})
    }
}
```

Следуя пункту 1.3 мы создали новую функцию `greetings` для улучшения читаемости и подняли отслеживание состояния нажатия кнопки продолжения в родительскую функцию, и с помощью добавления параметра к функции `OnboardingScreen`, сделали так, чтобы мы могли изменить состояние в `MyApp`. Таким образом `OnboardingScreen` теперь не изменяет состояние, а уведомляет `MyApp` о том, что кнопка была нажата и `MyApp` уже изменяет состояние.

## 1.7. Ленивые списки и rememberSaveable.

LazyColumn и LazyRow используются для отображения прокручиваемого столбца, при этом приложение отображает только видимые элементы на экране, таким образом не все элементы изначально загружаются на экран, а создаются в процессе прокрутки, тем самым обеспечивая хорошую производительность.

Листинг 1.7 LazyColumn.

```
@Composable
private fun Greetings(names: List<String> = List(1000) { "Sit" } ) {
    LazyColumn(modifier = Modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name)
        }
    }
}
```

LazyColumn и LazyRow эквивалентны RecyclerView в представлениях Android.

Remember функция работает только до тех пор , как компонуемы хранится в композиции . При повороте все действия перезапускаются, поэтому все состояния теряются. Это также происходит при любом изменении конфигурации и при смерти процесса. Чтобы этого не происходило нужно использовать rememberSaveable, который сохранит состояние даже при смене конфигурации.

## 1.8. Анимация.

В Compose есть несколько способов анимировать пользовательский интерфейс: от высокоуровневых API для простых анимаций до низкоуровневых методов для полного контроля и сложных переходов. Улучшим анимацию изменения размера при помощи функции animateDpAsState

Листинг 1.8 animateDpAsState.

```
@Composable
private fun Greeting(name: String) {

    var expanded by remember { mutableStateOf(false) }

    val extraPadding by animateDpAsState(
        if (expanded) 48.dp else 0.dp
    )
    Surface(
        color = MaterialTheme.colors.primary,
        modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
    ) {
        Row(modifier = Modifier.padding(24.dp)) {
            Column(modifier = Modifier
                .weight(1f)
                .padding(bottom = extraPadding))
            ) {
                Text(text = "Hello, ")
                Text(text = name)
            }
            OutlinedButton(
                onClick = { expanded = !expanded })
            ) {
                Text(if (expanded) "Show less" else "Show more")
            }
        }
    }
}
```

animateDpAsState возвращает State объект, значение value которого будет обновляться до тех пор пока анимация не прекратится. Эта анимация зависит от состояния expanded. Если же повторно нажать на кнопку во время раскрытия анимации, то она прерывается и указывает на новое значение и сворачивается обратно, не закончим анимацию разворота.

## 1.9. Стилизация приложения.

Посмотрим, что находится внутри файла ui/Theme.kt.

Листинг 1.9.1 Theme.kt.

```
package com.example.basicscodelab.ui.theme

import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material.MaterialTheme
import androidx.compose.material.darkColors
import androidx.compose.material.lightColors
import androidx.compose.runtime.Composable

private val DarkColorPalette = darkColors(
    primary = Purple200,
    primaryVariant = Purple700,
    secondary = Teal200
)

private val LightColorPalette = lightColors(
    primary = Purple500,
    primaryVariant = Purple700,
    secondary = Teal200
)

@Composable
fun BasicsCodelabTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable() () -> Unit
) {
    val colors = if (darkTheme) {
        DarkColorPalette
    } else {
        LightColorPalette
    }

    MaterialTheme(
        colors = colors,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}
```

Мы видим, что что BasicsCodelabTheme использует MaterialTheme в его реализации. MaterialTheme- это компонуемая функция, которая отражает принципы стилизации из спецификации материального дизайна .

Посколько MyApp использует данную тему, из любого его потомка мы можем получить три свойства MaterialTheme.

### Листинг 1.9.2 Использование свойств MaterialTheme.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                MyApp()
            }
        }
    }

    @Composable
    private fun Greeting(name: String) {
        var expanded by remember { mutableStateOf(false) }

        val extraPadding by animateDpAsState(
            if (expanded) 48.dp else 0.dp
        )
        Surface(
            color = MaterialTheme.colors.primary,
            modifier = Modifier.padding(vertical = 4.dp, horizontal = 8.dp)
        ) {
            Row(modifier = Modifier.padding(24.dp)) {
                Column(modifier = Modifier
                    .weight(1f)
                    .padding(bottom = extraPadding))
                ) {
                    Text(text = "Hello, ")
                    Text(text = name, style = MaterialTheme.typography.h4)
                }
                OutlinedButton(
                    onClick = { expanded = !expanded })
                ) {
                    Text(if (expanded) "Show less" else "Show more")
                }
            }
        }
    }
}
```

Мы можем создать свой стиль для текста или использовать как в примере выше стиль определенный темой, используя MaterialTheme.typography, что является предпочтительнее.

В целом, гораздо лучше хранить цвета, формы и стили шрифтов внутри файла MaterialTheme. Например, темный режим будет трудно реализовать,

если вы жестко запрограммируете цвета, и для его исправления потребуется много работы, подверженной ошибкам. Однако если все же нужно изменить цвет или стиль, то следует использовать *copy*.

```
Text(  
    text = name,  
    style = MaterialTheme.typography.h4.copy(  
        fontWeight = FontWeight.ExtraBold  
    )  
)
```

Тем самым мы лишь немного изменим его стиль, и не нужно будет думать о слишком больших проблемах в последующем.

Также мы можем определять новые цвета в файле Color.kt и устанавливать их для светлой/темной темы.

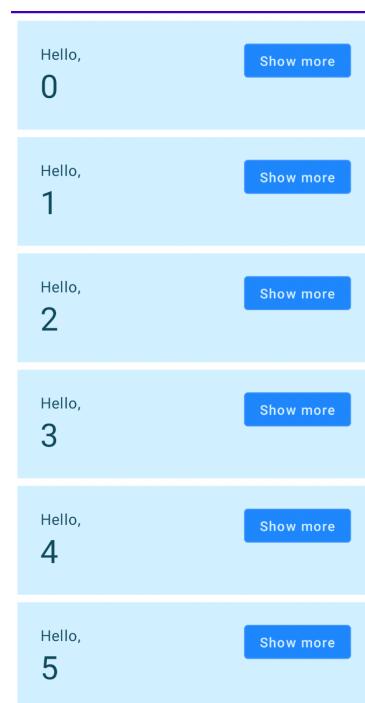


Рис 1.9 Приложение с изменёнными цветами

## Задание №2

При переходах между различными Activity пользователь всегда может вернуться на предыдущую, закрытую Activity при нажатии кнопки back на устройстве. Подобная логика реализована с помощью стека (Activity Stack).

Он реализован по принципу “последний зашел – первый вышел”.

При открытии новой Activity она становится вершиной, а предыдущая уходит в режим stop. Стек не может перемешиваться, он имеет возможность добавления на вершину новой Activity и удаление верхней текущей. Одна и та же Activity может находиться в стеке, сколько угодно раз.

Task — это набор Activity. Каждый таск содержит свой стек. В стандартной ситуации, каждое приложение имеет свой таск и свой стек.

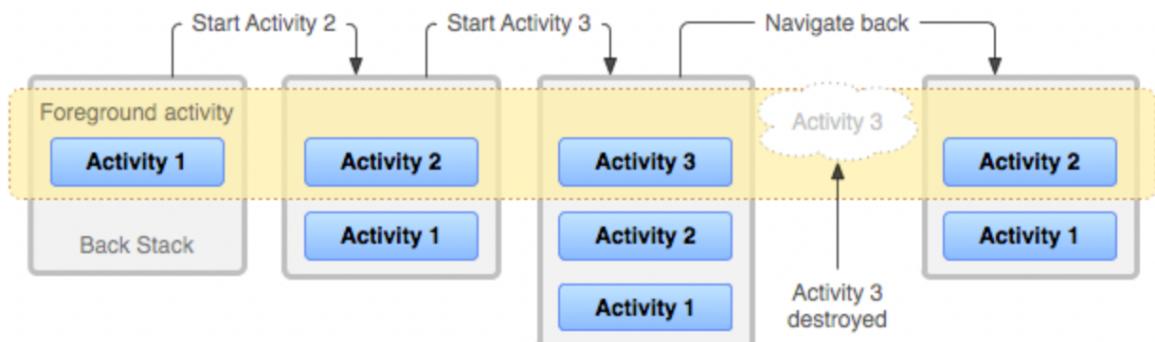


Рис 2.1 Алгоритм работы backstack.

Рассмотрим как можно работать с нашими активити так, чтобы они не засоряли backstack.

В этом пункте мы будем работать с активити при помощи методов `startActivity`, `startActivityForResult`, `setResult`, `onActivityResult`, `finish`.

*startActivity(Intent)* – создает новую активити, которая располагается в вершине нашего стека, аргумент Intent представляет собой задачу, которую необходимо выполнить приложению, в нашем случае переход к нужной нам Activity.

*startActivityForResult(Intent, Int requestCode)* – также создает новую активити, но имеет еще дополнительный аргумент requestCode используется для обработки результата в методе onActivityResult, допустим, если переход из активити возможен несколькими вариантами, будет понятно, откуда был получен результат.

*onActivityResult(Int requestCode, Int resultCode, Intent data)* – обрабатывает результат работы активити, вызванного startActivityForResult, о первом аргументе было сказано выше, второй аргумент является является результатом работы функции setResult в вызванной activity, data – это данные переданные из открытого activity.

*setResult(int resultCode, Intent data)* – возвращает результат, запустившему его активити.

*finish()* – завершает работу Activity

В данной пункте лабораторной работы нужно реализовать приложение с четырьмя активити с переходом между ними.

- FirstActivity -> SecondActivity
- SecondActivity -> ThirdActivity
- SecondActivity -> FirstActivity (При этом SecondActivity удаляется из backstack)
- ThirdActivity -> SecondActivity (ThirdActivity удаляется из backstack)

- ThirdActivity -> FirstActivity (ThirdActivity и SecondActivity удаляется из backstack)

Также по заданию нужно было реализовать bottom navigation меню для перехода в ActivityAbout из любого Activity.

Для перехода из первого activity был написан следующий код:

Листинг 2.1 Переход во второе Activity

```
private fun toSecond() {
    startActivity(Intent(this, SecondActivity::class.java))
}
```

Мы просто запускаем второе активити при помощи метода startActivity

Далее посмотрим на переходы второго активити:

Листинг 2.2 Переходы SecondActivity

```
private val requestCode = 1;

private fun toFirst() {
    finish()
}

private fun toThird() {
    startActivityForResult(Intent(this, ThirdActivity::class.java),
requestCode)
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (resultCode == Activity.RESULT_OK) finish()
}
```

В случае перехода в первое активити мы просто используем метод finish() для закрытия этого активити. Для перехода же в третье активити мы используем метод startActivityForResult, это необходимо, так как при переходе из третьего активити в первое мы должны будем закрыть и третью и второе активити и благодаря методу startActivityForResult мы

обрабатываем переданное значение методом setResult в onActivityResult, если было передано значение RESULT\_OK, мы закрываем вторую активити.

Листинг 2.3 Переходы ThirdActivity

```
private fun toFirst() {
    this.setResult(RESULT_OK)
    finish()
}

private fun toSecond() {
    finish()
}
```

В случае возвращения ко второму активити мы просто закрываем текущую, при переходе в первую мы также завершаем текущую активити, но также передаем результат выполнения выполнения второму активити, чтобы при помощи метода onActivityResult оно также было закрыто.

Также было реализовано bottom navigation меню для перехода в AboutActivity, для каждого из шаблонов activity, для этого в каждом layout файле был прописан следующий код:

Листинг 2.4 Переходы bottom navigation menu

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/toAbout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/design_default_color_primary_dark"
    app:itemTextColor="@color/white"
    android:foregroundGravity="center"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:menu="@menu/bottom_menu"/>
```

В атрибутах заданы цвет текста элементов меню, цвет заднего фона, его расположение относительно экрана и сами элементы меню, реализованные в соответственном ресурсе menu.

#### Листинг 2.4 Ресурс menu

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/about"
        android:title="@string/to_about"/>
</menu>
```

Также в каждом из activity был описан код для перехода в AboutActivity. С помощью метода setOnNavigationItemReselectedListener мы отслеживаем нажатие на элемент нашего меню и переходим к нему с помощью функции toAbout.

#### Листинг 2.4 toAbout

```
private fun toAbout(item: MenuItem) {
    when (item.itemId) {
        R.id.about ->startActivity(Intent(this,ActivityAbout::class.java))
    }
}
```

В этой функции определено действие, которое должно вызываться, при нажатии одного из пунктов в меню, в нашем случае в нашем меню только один пункт.

Для проверки правильности переходов между активити мы использовали Android Debug Bridge.

```
1|emulator64_arm64:/ $ dumpsys activity activities | grep 'Hist #' | grep 'com.example.lab2'
    * Hist #0: ActivityRecord{7d7bbbb8 u0 com.example.lab2/.MainActivity t39}
emulator64_arm64:/ $ dumpsys activity activities | grep 'Hist #' | grep 'com.example.lab2'
    * Hist #1: ActivityRecord{4a830ea u0 com.example.lab2/.SecondActivity t39}
    * Hist #0: ActivityRecord{7d7bbbb8 u0 com.example.lab2/.MainActivity t39}
emulator64_arm64:/ $ dumpsys activity activities | grep 'Hist #' | grep 'com.example.lab2'
    * Hist #2: ActivityRecord{3364db5 u0 com.example.lab2/.ThirdActivity t39}
    * Hist #1: ActivityRecord{4a830ea u0 com.example.lab2/.SecondActivity t39}
    * Hist #0: ActivityRecord{7d7bbbb8 u0 com.example.lab2/.MainActivity t39}
emulator64_arm64:/ $ dumpsys activity activities | grep 'Hist #' | grep 'com.example.lab2'
    * Hist #0: ActivityRecord{7d7bbbb8 u0 com.example.lab2/.MainActivity t39}
emulator64_arm64:/ $ dumpsys activity activities | grep 'Hist #' | grep 'com.example.lab2'
    * Hist #3: ActivityRecord{7651909 u0 com.example.lab2/.ActivityAbout t39}
    * Hist #2: ActivityRecord{8544086 u0 com.example.lab2/.ThirdActivity t39}
    * Hist #1: ActivityRecord{f99eeac u0 com.example.lab2/.SecondActivity t39}
    * Hist #0: ActivityRecord{7d7bbbb8 u0 com.example.lab2/.MainActivity t39}
emulator64_arm64:/ $ █
```

Рис. 2.2 Отображение переходов в терминале.

В ходе проверки двух одинаковых activity в стеке не находилось, следовательно работа сделана верно, однако android studio нас предупреждает о том, что метод OnActivityResult и метод startActivityForResult являются deprecated, что значит, что в последующих версиях android studio он будет удален, существует другой способ для получения результата работы этих двух методов.

⚠ 'startActivityForResult(Intent!, Int): Unit' is deprecated. Deprecated in Java :37  
⚠ 'onActivityResult(Int, Int, Intent?): Unit' is deprecated. Overrides deprecated member in 'androidx.activity.ComponentActivity'. Deprecated in Java :41

Рис. 2.3 Предупреждение android studio.

Чтобы решить эту проблему мы должны использовать activityResultLauncher

#### Листинг 2.5 Альтернативный переход в ThirdActivity

```
private var launcher: ActivityResultLauncher<Intent>? = null;

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivitySecondBinding.inflate(layoutInflater)
    setContentView(binding.root)

    launcher = registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
        result: ActivityResult ->
        if (result.resultCode == RESULT_OK) finish()
    }

    binding.toFirst.setOnClickListener {
        toFirst()
    }

    binding.toThird.setOnClickListener {
        toThird()
    }

    binding.toAbout.setOnNavigationItemReselectedListener { toAbout(it) }
}

private fun toThird() {
    launcher?.launch(Intent(this, ThirdActivity::class.java))
}
```

Мы используем класс ActivityResultLauncher , который запускает нашу новую активити. registerForActivityResult – “регистрирует” на результат с активити, которое будет запущено с этого launcher. Таким образом данный

метод начнет работать в тот момент, когда с активити, запущенного данным launcher придет результат, как и в прошлом случае с метода setResult.

Тем самым мы можем создать несколько таких лаунчеров, которые будут отвечать за запуск различных активити и к каждой активити будет привязан свой RegisterForActivityResult, выполняющий различные действия. Данный метод решения был также успешно протестирован при помощи adb.

### Что произойдёт, если не указать Activity в манифесте?

При переходе на данный Activity приложение будет вылетать.

## Задание №3

Решим предыдущую задачу без использования startActivityForResult, в качестве альтернативного решения мы будем использовать известный нам startActivity, в который мы будем передавать Intent с дополнительным флагом.

### Листинг 3.1 Альтернативный переход в ThirdActivity

```
private fun toThird() {
    startActivity(Intent(this, ThirdActivity::class.java))
}
```

### Листинг 3.2 Альтернативный переход в FirstActivity из ThirdActivity.

```
private fun toFirst() {
    val intent = Intent(this, MainActivity::class.java)
        .addFlags(FLAG_ACTIVITY_CLEAR_TOP)
    startActivity(intent)
}
```

Мы задаем дополнительный флаг к Intent, а именно флаг FLAG\_ACTIVITY\_CLEAR\_TOP – он используется таким образом, что ищет в таске создаваемое Activity и если находит, то закрывает все, что находилось выше него, таким образом при переходе к первой активити в

таске будет SecondActivity и ThirdActivity, которые будут успешно закрыты этим флагом. Данный метод решения был также успешно протестирован с помощью adb.

## Задание №4

Допустим мы хотим дополнить наше предыдущее приложение и с третьего Активити добавить возможность проходить психологический тест. Мы хотим сделать так, чтобы по мере прохождения теста, переходя к следующему вопросу, пользователь не мог вернуться на вопрос назад, нажатием соответствующей кнопки на экране и изменить ответ на предыдущий вопрос и ему необходимо было бы начать сначала. Для решения данной задачи мы будем использовать флаг Intent FLAG\_ACTIVITY\_NO\_HISTORY.

Листинг 4.1 Начало психологического теста.

```
private fun startTest() {  
    val intent = Intent(this, PsychoTestFirst::class.java)  
        .addFlags(Intent.FLAG_ACTIVITY_NO_HISTORY)  
    startActivity(intent)  
}
```

Листинг 4.2 Переход к следующему вопросу.

```
private fun nextQuestion() {  
    val intent = Intent(this, PsychoTestSecond::class.java)  
        .addFlags(Intent.FLAG_ACTIVITY_NO_HISTORY)  
    startActivity(intent)  
}
```

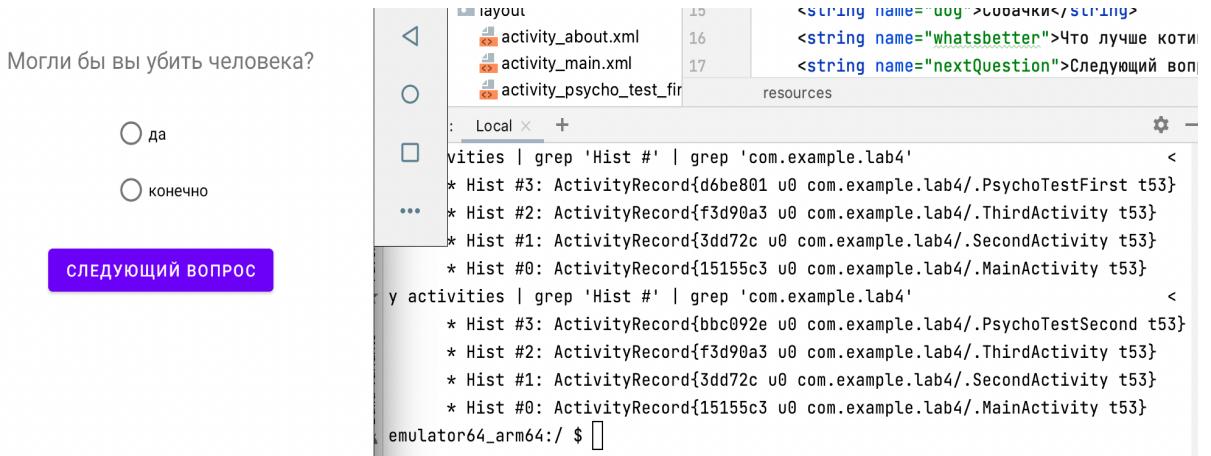


Рис. 4.1 Отображение переходов в терминале после перехода ко второму вопросу.

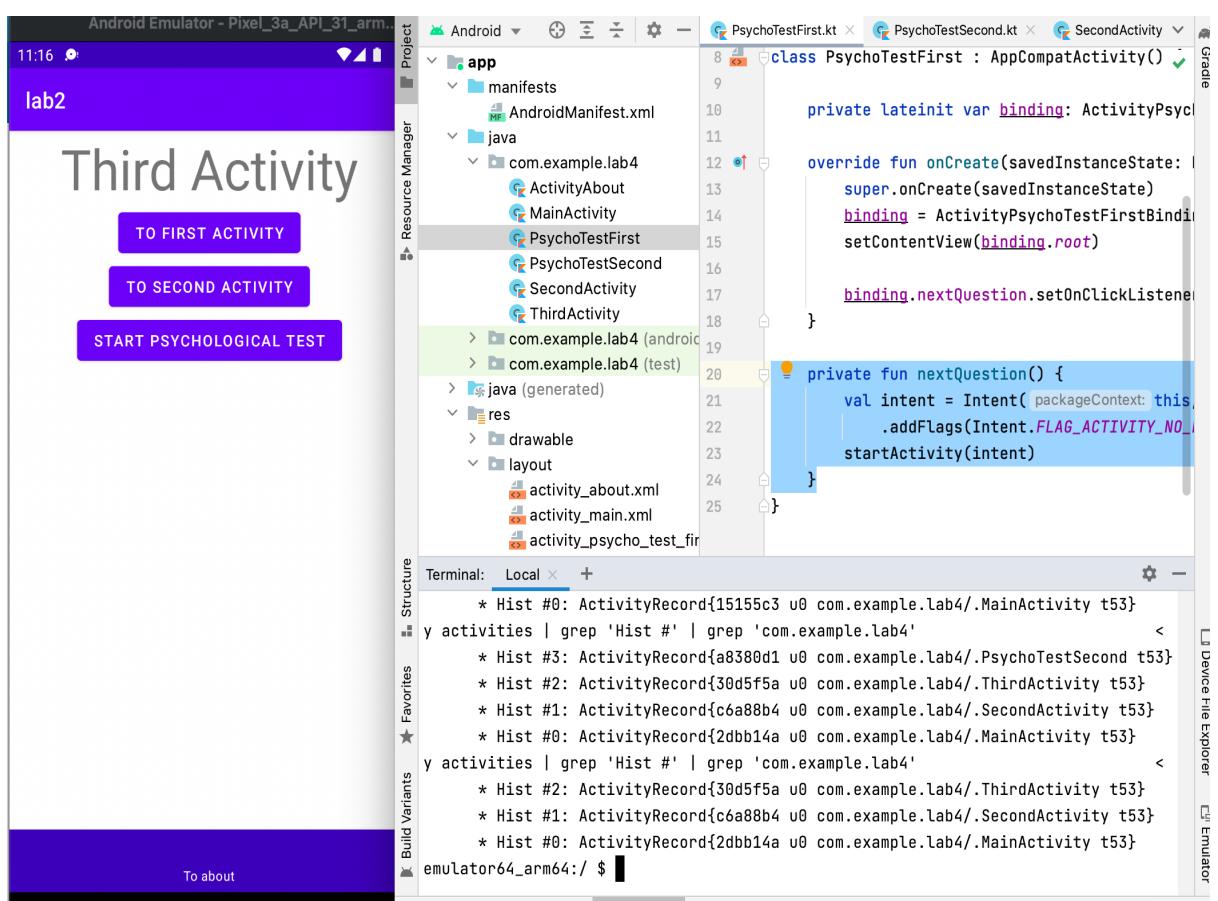


Рис. 4.1 Отображение переходов после нажатия кнопки назад.

`FLAG_ACTIVITY_NO_HISTORY` – позволяет сделать так, что бы при переходе на Activity оно не сохранялось в таске, таким образом после перехода ко второму вопросу, активити с первым вопросом не было сохранено и после нажатия кнопки назад мы возвращаемся к третьему

активити и вынуждены проходить тест заново, как изначально и задумывали.

## Задание №5

Фрагмент представляет кусочек визуального интерфейса приложения, который может использоваться повторно и многократно. Фрагмент существует в контексте activity и имеет свой жизненный цикл, вне activity обособлено он существовать не может. Каждая activity может иметь несколько фрагментов.

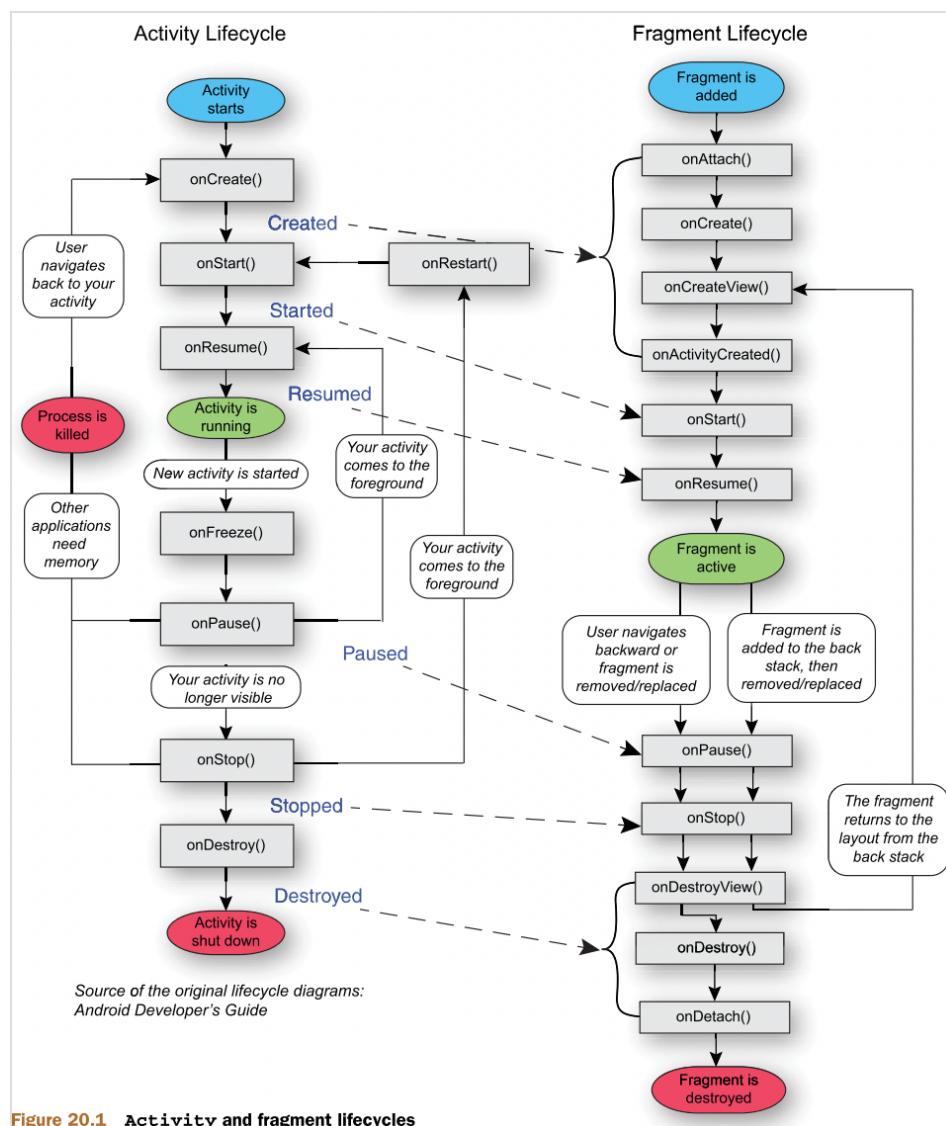


Figure 20.1 Activity and fragment lifecycles

## Рис 5.1 Жизненный цикл фрагментов.

Для взаимодействия между фрагментами мы будем использовать Navigation Graph, который позволяет упростить реализацию навигации между экранами назначения в приложении. Данный ресурс содержит информацию о направлениях перехода.

Для начала создадим три фрагмента, наследовавшись от класса Fragment, у каждого фрагмента есть свой ресурс разметки, для его отображения нужно указать его в методе onCreateView.

### Листинг 5.1 Код первого фрагмента FirstFragment

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    val binding = FragmentFirstBinding.inflate(inflater)
    return binding.root
}
```

Аналогичный образом опишем код для остальных трех фрагментов.

Далее необходимо создать навигационный граф, указав в нем все переходы между фрагментами с помощью редактора навигации, в который мы добавляем фрагменты и переходы между ними, соединяя их между собой. Для использования навигации нужно добавить следующие зависимости в build gradle приложения.

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

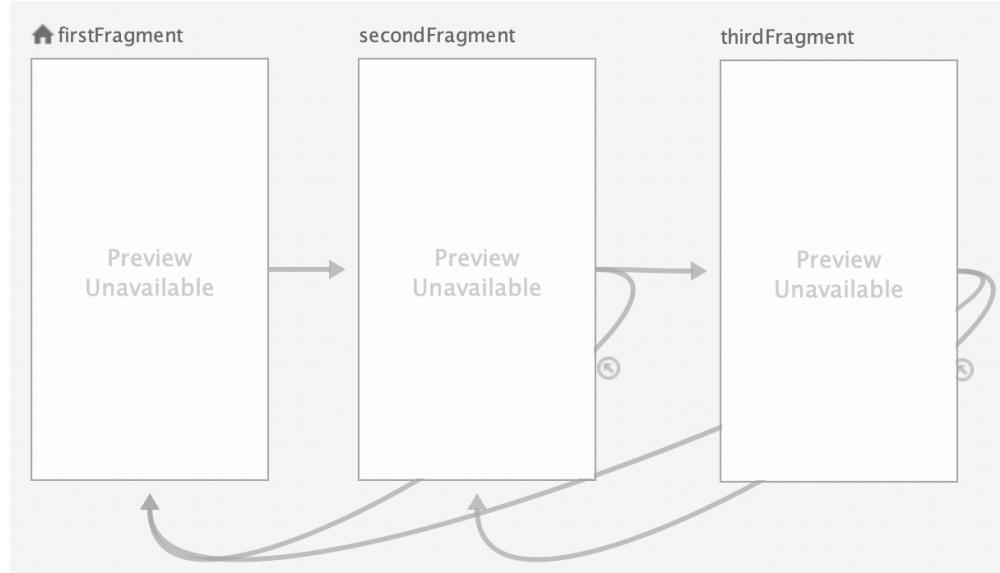


Рис 5.2 Редактор навигации.

Листинг 5.2 часть кода navigation graph.

```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/navigation_graph"
    app:startDestination="@+id/firstFragment">

    <fragment
        android:id="@+id/firstFragment"
        android:name="com.example.lab5.FirstFragment"
        android:label="FirstFragment" >
        <action
            android:id="@+id/action_firstFragment_to_secondFragment"
            app:destination="@+id/secondFragment" />
    </fragment>
```

Здесь мы видим, что устанавливается начальный фрагмент `firstFragment`, при помощи `app:startDestination` и дальше описаны все фрагменты и переходы между ними, направления перехода задается при помощи `app:destination`.

Далее необходимо описать `Activity`, из которого будут запускаться фрагменты. Для этого в ресурсе разметки этого `Activity` нужно добавить `FragmentContainerView`.

### Листинг 5.3 FragmentContainerView.

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"

    app:defaultNavHost="true"
    app:navGraph="@navigation/navigation_graph" />
```

androidName – содержит имя класса контейнера фрагмента, в котором NavController будет отображать фрагменты.

app:defaultNavHost - отвечает за перехват системной кнопки Back

app:navGraph – указывает на ресурс навигационного графа.

Теперь остается только добавить обработчик для кнопок, для перехода между фрагментами. Для этого нам нужен navController, который мы можем получить при помощи метода findNavController().

После получения NavController, используем его метод navigate() чтобы перейти к экрану назначения. Метод navigate() принимает ID ресурса.

Также добавим обработчик для нажатия по кнопкам bottomNavigation меню.

### Листинг 5.4 Добавление обработчиков.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    val binding = FragmentFirstBinding.inflate(inflater)

    val navController = findNavController()

    binding.toSecond.setOnClickListener {
        navController.navigate(R.id.action_firstFragment_to_secondFragment)
    }

    binding.toAbout.setOnNavigationItemReselectedListener {
        when(it.itemId) {
            R.id.about -> navController.navigate(R.id.global_about)
        }
    }

    return binding.root
}
```

Для перехода к activityAbout мы также используем navController. Для этого мы в файле навигации используем Global Action. Он используется для того, чтобы создать общее действие, которое могут использовать несколько мест назначения (фрагментов), в нашем случае таким действием как раз и

является переход к ActivityAbout. Код объявления global action в ресурсе навигации выглядит следующим образом:

```
<action android:id="@+id/global_about"
    app:destination="@+id/aboutActivity"/>

<activity
    android:id="@+id/aboutActivity"
    android:name="com.example.lab5.ActivityAbout"
    android:label="AboutActivity" />
```

Так делаем для всех фрагментов и, по сути, приложение почти готово. Почти, так как по заданию необходимо сделать так, чтобы при нажатие кнопки back в первом фрагменте, приложение всегда закрывалось, но сейчас этого не происходит, вследствие того, что у фрагментов также есть свой backstack. Для того, чтобы его контролировать мы можем использовать следующие два атрибута внутри ресурса навигации в теге action:

Листинг 5.5 Атрибуты для контроля backstack фрагментов.

```
<fragment
    android:id="@+id/secondFragment"
    android:name="com.example.lab5.SecondFragment"
    android:label="SecondFragment" >
    <action
        android:id="@+id/action_secondFragment_to_thirdFragment"
        app:destination="@+id/thirdFragment" />
    <action
        android:id="@+id/action_secondFragment_to_firstFragment"
        app:destination="@+id/firstFragment"
        app:popUpTo="@+id/firstFragment"
        app:popUpToInclusive="true" />
</fragment>
```

App:popUpTo – используется для указания до какого фрагмента вытокнуть другие фрагменты с верхушки стека.

App:popUpToInclusive="true" – Указание на то, что необходимо удалить дубликаты из backstack.

Определим эти атрибуты для необходимых переходов и все, теперь наше приложение работает корректно и backstack не захламляется.

## **Вывод**

В первой части лабораторной работы был рассмотрен Jetpack Compose. Данный инструмент для создания UI показался мне очень удобным, его использование, как мне кажется, намного быстрее, чем использование обычных методов для этих задач. В последующих пунктах мы работали с взаимодействием между различными Activity и рассмотрели, что такое backstack. Были рассмотрены несколько способов контроля backstack, чтобы в нем не было дубликатов одной активности. Было также предложено альтернативное использование startActivityForResult и onActivityResult в связи с тем, что они deprecated. В последнем пункте работы мы поработали с Fragment, обеспечив навигацию между ними при помощи navigation graph, позволяющего описать переходы между фрагментами.

## **Время на выполнение работы**

Задача №1 ~ 7 часов

Задача №2 ~ 6 часов

Задача №3 ~ 1.5 часа

Задача №4 ~ 40 минут

Задача №5 ~ 6 часов